EDAF30 – Programmering i C++

Resurshantering. Objektorientering. Klasser.

Sven Gestegård Robertz
Datavetenskap, LTH

2015

Innehåll

Typdeklarationer med `typedef` och `using`

`typedef`

```
typedef unsigned long ulong;
typedef char rad[80];
typedef double (*funpek)(int);
```

`using` (C++ 11)

```
using ulong = unsigned long;
using rad = char[80];
using funpek = double (*)(int);
```

Användning

Följande deklarationer är ekvivalenta:

```
ulong a[12];      unsigned long a[12];
rad namn;         char namn[80];
funpek pf;        double (*pf)(int);
```


Typomvandlingar (casting)
Implicita Typomvandlingar

Automatiska typomvandlingar

- Uttryck av typen $x \odot y$, för någon binär operator $\odot$
  Ex: `double + int ==> double`
  `float + long + char ==> float`
- Tilldelningar och initieringar: Värdet i högerledet konverteras till samma datatyp som i vänsterledet
- Konvertering av typen hos aktuella parametrar till typen för de formella parametrarna
- Villkor i `if`-satser, etc. $\Rightarrow$ **`bool`**
- C-array $\Rightarrow$ pekare (*array decay*)
- `0` $\Rightarrow$ `nullptr` (tom pekare, i C++11, tidigare definierade man ofta konstanten `NULL`)

Typomvandlingar (*casting*)
Explicita typomvandlingar, C-stil

Syntax

```
(typnamn)uttryck; // (Både i C och i C++, som i Java)
typnamn(uttryck); // (Alternativ syntax i C++)
```

För varianten `typnamn(uttryck)` måste `typnamn` vara *ett ord*, d v s `int *(...)` eller `unsigned long(...)` är inte OK.

Exempel

```
char *p = (char *) 07650; // heltal ==> pekare
long i = (long) p;        // pekare ==> heltal

int x=3, y=4;
double a = ((double)(x+y))/2;

unsigned char a = (unsigned char) -3;            //  253
signed char b   = (signed char) 1000;            // -24   (3)e8
unsigned c      = (unsigned) (signed char) 1000; //  4294967272
```


Typomvandlingar (*casting*)
Namngivna typomvandlingar

- **`static_cast`**`<new_type> (expr)`
  - omvandlar mellan kompatibla typer (*kollar inte talområden*)
- **`reinterpret_cast`**`<new_type> (expr)`
  - inget skyddsnät, samma som C-stil
- **`const_cast`**`<new_type> (expr)` - lägger till eller tar bort **`const`**
- **`dynamic_cast`**`<new_type> (expr)` - används för pekare till klasser. Gör typkontroll vid *run-time*, som i Java.

Exempel

```
char c;              // 1 byte
int *p = (int*) &c;  // pekar på int: 4 bytes

*p = 5; //fel vid exekvering, stack-korruption

int *q = static_cast<int*> (&c); // kompileringsfel
```

Minnesallokering

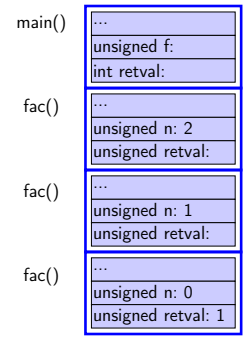
Två sorters minnesallokering:

- på *stacken* - *automatiska* variabler. Förstörs när programmet lämnar det *block* där de deklarerats.
- på *heapen* - *dynamiskt allokerade* variabler. Överlever tills de explicit avallokeras.

Minnesallokering
Exempel: allokering på *stacken*

```
unsigned fac(unsigned n)
{
  if(n == 0)
    return 1;
  else return n * fac(n-1);
}

int main()
{
  unsigned f = fac(2);
  cout << f;
  return 0;
}
```



Minnesallokering
Dynamiskt minne, allokering "på *heapen*", eller "i *free store*"

Utrymme för dynamiska variabler allokeras med `new`

```
double* pd = new double;      // allokera en double
*pd = 3.141592654;            // tilldela ett värde
float* px;
float* py;
px = new float[20];           // allokera array
py = new float[20] {1.1, 2.2, 3.3};   // allokera och initiera
```

Minne frigörs med `delete`

```
delete pd;
delete[] px; // [] krävs för C-array
delete[] py;
```


Minnesallokering
Varning! se upp med parenteser

Allokering av `char[80]`

```
char* c = new char[80];
```

Nästan samma...

```
char* c = new char(80);
```

Nästan samma...

```
char* c = new char{80};
```

De två senare allokerar *en byte*

och *initierar* den med värdet 80 (`'P'`).

```
char* c = new char('P');
```


Minnesallokering
ägarskap för resurser

För dynamiskt allokerade objekt är *ägarskap* viktigt

- Ett objekt eller en funktion kan *äga* ett objekt
- *Ägaren* är ansvarig för att avallokera objektet
- Om du har en pekare måste du veta *vem som äger objektet den pekar på*
- Ägarskap kan *överföras* vid funktionsanrop
  - men måste inte
  - var tydlig

Varje gång du skriver **new** är du ansvarig för att någon kommer att göra **delete** *när objektet inte ska användas mer*.

Minnesallokering

Typiskt misstag: Att glömma allokera minne

```
char namn[80];

*namn = 'Z'; // Ok, namn allokerad på stacken. Ger namn[0]='Z'


char *p;      // Oinitierad pekare
              // Ingen varning vid kompilering

*p = 'Z';     // Fel! 'Z' skrivs till  en slumpvis vald adress

cin.getline(p, 80); // Ger (nästan) garanterat exekveringsfel
                    // ("Segmentation fault") eller
                    // minneskorruption
```

Minnesallokering

Exempel: misslyckad read_line

```
char* read_line() {
   char temp[80];
   cin.getline(temp, 80);
   return temp;
}

void exempel () {
        cout << "Ange ditt namn: ";
        char* namn = read_line();

        cout << "Ange din bostadsort: ";
        char* ort = read_line();

        cout << "Goddag " << namn << " från " << ort << endl;
}
```

"Dangling pointer": pekare till objekt som inte längre finns

Minnesallokering

Delvis korrigerad version av read_line

```
char* read_line() {
   char temp[80];
   cin.getline(temp, 80);
   char *res = new char[strlen(temp)+1];
   strcpy(res, temp);
   return res; // Dynamiskt allokerat överlever
}
void exempel () {
   cout << "Ange ditt namn: ";
   char* namn = read_line();
   cout << "Ange din bostadsort: ";
   char* ort = read_line();
   cout << "Goddag " << namn << " från " << ort << endl;
}
```

Fungerar, men minnesläcka !

Minnesallokering

Ytterligare korrigerad version av read_line

```
char* read_line() {
   char temp[80];
   cin.getline(temp, 80);
   char *res = new char[strlen(temp)+1];
   strcpy(res, temp);
   return res; // Dynamiskt allokerat överlever
}
void exempel () {
   cout << "Ange ditt namn: ";
   char* namn = read_line();  // Ta över ägarskap av sträng
   cout << "Ange din bostadsort: ";
   char* ort = read_line();
   cout << "Goddag " << namn << " från " << ort << endl;

   delete[] namn;     // Avallokera strängarna
   delete[] ort;
}
```


Minnesallokering
C++: Smarta pekare

I standardbiblioteket <memory> finns två "smarta" pekare:

- std::unique_ptr används om den är den *enda ägaren* till objektet
- std::shared_ptr används om ett objekt har *flera ägare*.

shared_ptr innehåller en *referensräknare*: när *sista* shared_ptr till ett objekt förstörs avallokeras objektet. Jfr. *garbage collection* i Java.

Minnesallokering
C++: Smarta pekare

Exempel

```
void example()
{
    unique_ptr<char[]> safe_c{new char[80]};
    char* c = safe_c.get();

    cin.getline(c,80);
    auto len = strlen(c);

    char last = safe_c[len-1] ; // indexering med [] fungerar

    cout << "last char: "<< last << endl;
}
```

- Genom att använda std::unique_ptr<char[]> så kommer automatiskt **delete**[] att göras när safe_c tas bort (i detta exempel: när funktionen returnerar).
- För att få en **char*** används unique_ptr::get().

Minnesallokering
C++: Smarta pekare

Tumregler för parametrar till funktioner:

om ägarskap inte ska överföras

- Använd "råa" pekare
- Använd std::unique_ptr<T> **const** &

om ägarskap ska överföras

.

- Använd *värdeanropad* std::unique_pointer<T> (då måste man använda std::move())
- Detta är bara en orientering om att smarta pekare finns.
- "Råa" pekare är så vanliga att ni måste lära er behärska dem först.

C++: Smarta pekare
Grov sammanfattning

"Råa" ("nakna") pekare:

- Programmeraren ansvarar för allt
- Risk för minnesläckor
- Risk för *dangling pointers*

Smarta pekare:

- Ingen (mindre) risk för minnesläckor
- (liten) Risk för *dangling pointers* om de används fel (t ex mer än en `unique_ptr`)

Objektorienterad programutveckling
Grundläggande begrepp

Objekt

- Representation av verkligt föremål eller företeelse
- Egenskaper - datamedlemmar (*attribut* i Java)
- Operationer - medlemsfunktioner (*metoder* i Java)

Klass

- Beskrivning av en viss typ av objekt
- Ett objekt sägs vara en *instans* av en klass

Objektorienterad programutveckling
Relationer mellan objekt

Association känner till
Bil —— Person (ägare)

Komposition har
Bil —— Motor

Generalisering är
Bil —— Fordon

```
class Car : public Vehicle {
  Person* owner;
  Engine e;
};
```

Notera

Till skillnad från i Java så kan ett objekt både *referera till* andra objekt och *innehålla* andra objekt.

Objektorienterad programutveckling
Objektorienterad programmering

Inkapsling (encapsulation, information hiding)

- Samla alla egenskaper på ett ställe
- Gömma implementationsdetaljer

Arv Återanvändning och specialisering

Polymorfism samma interface till flera olika typer

- *Virtuella* funktioner: *Subtyping*
- *Parametriserade* programenheter: *mallar* (*templates*)
- *Överlagring* av funktioner (*ad-hoc-polymorfism*)

Polymorfism
Tidig och sen bindning

Virtuella funktioner Betydelsen bestäms först *vid exekvering* (*dynamisk el. sen bindning* (*late binding*))

Parametriserade programenheter (generics, mallar i C++) Enheter där vissa av de ingående typerna är parametrar. *Ex.*: i `vector` är elementen av godtycklig typ. Heltalsvektor, t ex `vector<int>`. Detta bestäms *vid kompileringen* (*statisk bindning, early binding*)

Överlagring av funktioner Betydelsen hos en funktion bestäms av antal och typ av parametrarna. (*statisk binding*)

Klassdefinitioner

Klass Egendefinierad typ

Objekt Instans av en klass (notera att *objekt* även kan ha en generellare betydelse i C++, enligt tidigare definition)

Typisk klassdefinition:

```
class Foo {
public: // Den utåt synliga delen
    // Medlemsfunktioner (dekl.)
    // och ev. datamedlemmar
private: // Den dolda delen
    // Datamedlemmar och
    // ev. medlemsfunktioner
}; // Alltid semikolon efter klassdef.
```

- I C++ är **`struct`** samma som **`class`**, fast allting är **`public`** om ingen synlighet anges.

Klassdefinitioner
Deklarationer och definitioner av medlemsfunktioner

Medlemsfunktioner (⇔ metoder i Java)

Dekfinition av klass

```
class   Foo {
public:
    int fun(int, int);  // Deklaration av medlemsfunktion
    // ...
};
```

Obs! Semikolon efter klass

Definition av medlemsfunktion

```
//      Definition av funktionen:
int Foo::fun(int x, int y) {
    // ...
}
```

Inget semikolon efter funktion

Klassdefinitioner
Medlemsfunktioner och `inline`

Inline-definition av en funktion

- Koden läggs ut direkt på det ställe anropet görs (inget hopp till funktionskod på annat ställe)
- Lämpligt endast för mycket enkla funktioner
- Implicit om funktionsdefinitionen skrivs direkt i klassdefinitionen
- Om funktionen läggs utanför klassdefinitionen (via ::-notation) används nyckelordet `inline` framför

Klassdefinitioner
Medlemsfunktioner och `inline`, exempel

Inline i klassdefinitionen:

```
class Foo {
  public:
    int getValue() {return value;}
  // ...
};
```

Inline utanför klassdefinitionen:

```
inline int Foo::getValue()
{
    return value;
}
```


Filstruktur för klasser

Klassdefinitionen läggs i en headerfil (.h eller .hpp)

- För att inte riskera att definiera en klass mer än en gång används direktiv:

```
#ifndef FOO_H
#define FOO_H
//...
class Foo {
//...
};
#endif

// I Visual Studio finns även:
#pragma once
```

Medlemsfunktioner läggs i en källkodsfil (.cpp)

Konstruktorer
Initiering av medlemsvariabler

Initiering sker i en *konstruktor*, en speciell medlemsfunktion med samma namn som klassen (som i Java)

Konstruktorer
Två sätt att skriva initiering av medlemmar

Med initierarlista i konstruktorn.

```
class Bar {
private:
    int value;
    bool flag;
public:
    Bar(int v, bool b) :value{v},flag{b} {}
};
```

Medlemmar kan ha en *default initializer*, i C++11:

```
class Foo {
public:
    Foo() {}
    Foo(int v) :value{v} {}
private:
    int value{0}; // C++11
};
```

Om en medlem både har *default initializer* och initierare i konstruktorn finns, används initieraren i konstruktorn.

Konstruktorer

Defaultkonstruktor

- Definieras automatiskt om det inte finns någon konstruktor alls definierad (explicit: `=default`, kan ej anropas om `=delete`)
- Svarar mot en konstruktor utan parametrar
- Får ha parametrar med defaultvärden

Defaultkonstruktor med initierarlista.

```
class Bar {
public:
  Bar(int v=100, bool b=false) :value{v},flag{b} {}
private:
    int value;
    bool flag;
};
```

Anrop av defaultkonstruktor kan *inte göras med tomma parenteser*.

Konstruktorer
Defaultkonstruktor och parenteser

```
Bar b1;
Bar b2{};
Bar be();   // kompileringsfel!  "most vexing parse"
Bar b3(25); // OK

Bar* bp1 = new Bar;
Bar* bp2 = new Bar{};
Bar* bp3 = new Bar(); //OK
```


Konstruktorer
Initiering och tilldelning

Man *kan* skriva som i Java, men det är inte lika effektivt.

Som i Java: tilldelning i konstruktor

```
class Foo {
public:
    Foo(int v) {
        value = v;  //OBS! tilldelning, inte initiering
   }
private:
    int value; // initieras innan konstruktorns kropp exekveras
};
```


Konstruktorer

Kopieringskonstruktor

- Anropas vid t.ex. initiering av ett objekt då detta ska bli en kopia av ett annat objekt av samma typ
- Anropas *inte* vid tilldelning
- Kan definieras, men i annat fall definieras automatiskt en standardvariant av kopieringskonstruktor (`=default`, `=delete`)

```
Bar b1(10,false};

Bar b2{b1};   // kopieringskonstruktorn anropas
Bar b3(b2);   // kopieringskonstruktorn anropas
Bar b4 = b2;  // kopieringskonstruktorn anropas

b4 = b3;      // kopieringskonstruktorn anropas inte
```


Konstruktorer

Kopieringskonstruktor

- Har en **const** & som parameter: `Bar::Bar(const Bar& b);`

Typomvandlingskonstruktor

- Anropas då konstruktorn anropas med uttryck av annan typ

```
class KomplextTal {
public:
    KomplextTal():re(1),im(0) {}        //Default
    KomplextTal(float x):re(x),im(0) {} //Typomv.
    KomplextTal(const KomplextTal& k) :re{k.re},im{k.im} {}
    //...
private:
    float re, im;
};
```


Typomvandlingskonstruktor och `explicit`

```
struct A
{
    A(int) {}
    // ...
};

struct B
{
    explicit B(int) {}
    // ...
};
```

```
A a1(2);      // OK      B b1(2);      // OK
A a2 = 1;     // OK      B b2 = 1;     // Fel! [1]
A a3 = (A)1;  // OK      B b3 = (B)1;  // OK: explicit cast
```

```
[1]: conversion from 'int' to non-scalar type 'B' requested
```

explicit gör att en konstruktor inte kan anropas implicit för typomvanding.

Konstruktorer
Delegering (C++11)

I C++11 kan en konstruktor anropa en annan (som `this(...)` i Java).

```
struct Test{
    int val;
    bool bar;

    Test(int v, bool b) :val{v},bar{b} {}

    Test(int v) :Test(v,true) {};      // delegering

    Test() :Test(0,false) {};           // delegering
};
```


Destruktorer

- När minnesutrymme allokeras dynamiskt inne i en konstruktor (med `new`) måste detta avallokeras igen så småningom.
- Det finns en "motsats" till konstruktor som anropas vid avallokering (t.ex. vid `delete`).
- Denna kallas *destruktor* (namnet inleds med komplementoperatorn ~ ("tilde")

```
class Foo {
public:
    // ...
    ~Foo(); // Dekl. av destruktor
};
```


Exempel: Vektor

Klassdefinition

```
class Vektor {
public:
    Vektor(int lng = 10);    // Även defaultkonstruktor
    Vektor(const Vektor& v); // Kopiering!
    ~Vektor(); // Destruktor
    int lngd() {return ant;} // Inline!
    int get_elem(int ind);
    void set_elem(int ind, int val);
private:
    int *p; // Själva vektorn
    int ant;// Antalet element
};
```


Exempel: Vektor

Konstruktorer

```
Vektor::Vektor(int lng) : ant(lng) {
    assert(ant>=0);     // Indexkontroll  (unsigned vore bättre)
    p = new int[ant];   // Dynamisk allokering
}

//Kopieringskonstruktor
Vektor::Vektor(const Vektor& v) : ant(v.ant) {
    p = new int[ant];
    for (int i=0; i<ant; i++)
        p[i] = v.p[i];
}
```


Exempel: Vektor

Övriga medlemsfunktioner

```
int Vektor::get_elem(int ind) {
    assert(ind>=0 && ind<ant);
    return p[ind];
}
void Vektor::set_elem(int ind, int val) {
    assert(ind>=0 && ind<ant);
    p[ind] = val;
}
```


Exempel: Vektor

Destruktorn

```
//Deallokera allt dynamiskt minnesutrymme
Vektor::~Vektor() {
    delete[] p;
}
```

Destruktorn kan även skrivas inne i klassdefinitionen:

```
// ...
   ~Vektor() {delete[] p;}
// ...
```

Exempel: Vektor
Resurshantering

Exempel: objekt på stack och heap i ett block

```
{
    Vektor v;    // Konstruktor anropas (stack-allokering)
    Vektor *pv;
    pv = new Vektor; // Konstruktoranrop (heap-allokering)

    // ...

    delete pv; // Destruktor anropas för objektet som pv pekar på

    // ...

}// Destruktor anropas för v
```


Sammanfattning

Vi har talat om

- Resurshantering
  (minnesallokering, minnesläckor, dangling pointers)
- Klasser

Nästa föreläsning:

Vi kommer att introducera

- **const** för objekt och medlemmar
- Statiska medlemmar
- Pekaren **this**
- Vänner (**friend**)
- Överlagring av operatorer